# Compiler Directed Network-on-Chip Reliability Enhancement for Chip Multiprocessors *

Ozcan Ozturk
Bilkent University, Turkey
ozturk@cs.bilkent.edu.tr

Mahmut Kandemir, Mary J. Irwin
Pennsylvania State University, USA
{kandemir,mji}@cse.psu.edu

Sri H. K. Narayanan †
Pennsylvania State University, USA
snarayan@cse.psu.edu

## Abstract

Chip multiprocessors (CMPs) are expected to be the building blocks for future computer systems. While architecting these emerging CMPs is a challenging problem on its own, programming them is even more challenging. As the number of cores accommodated in chip multiprocessors increases, network-on-chip (NoC) type communication fabrics are expected to replace traditional point-to-point buses. Most of the prior software related work so far targeting CMPs focus on performance and power aspects. However, as technology scales, components of a CMP are being increasingly exposed to both transient and permanent hardware failures.

This paper presents and evaluates a compiler-directed power-performance aware reliability enhancement scheme for network-on-chip (NoC) based chip multiprocessors (CMPs). The proposed scheme improves on-chip communication reliability by duplicating messages traveling across CMP nodes such that, for each original message, its duplicate uses a different set of communication links as much as possible (to satisfy performance constraint). In addition, our approach tries to reuse communication links across the different phases of the program to maximize link shutdown opportunities for the NoC (to satisfy power constraint). Our results show that the proposed approach is very effective in improving on-chip network reliability, without causing excessive power or performance degradation. In our experiments, we also evaluate the performance oriented and energy oriented versions of our compiler-directed reliability enhancement scheme, and compare it to two pure hardware based fault tolerant routing schemes.

***Categories and Subject Descriptors*** D.3.4 [*Programming Languages*]: Processors—compilers, memory management, optimization

***General Terms*** Experimentation, Management, Design, Performance

***Keywords*** Chip multiprocessors, reliability, NoC, Compiler

## 1. Introduction

As processor design has become severely power and performance limited, it is now commonly accepted that staying on the current performance trajectory (doubling of chip performance every 24 to 36 months) will come about through the integration of multiple processors (cores) on a chip rather than through increases in the clock rate of single processors. Several chip manufacturers already have dual core chips on the market (e.g., Intel's dual core Montecito [28], the dual core AMD Athlon [2]), with more aggressive configurations being delivered or prototyped (e.g., Sun's eight core Niagara [23], IBM's Cell [20], Intel's quad core Xeon [19], and Intel's 80 core TeraFlop [18]). In the long run, one can expect the number of cores in chip multiprocessors (CMPs) to increase.

It is expected that CMPs will be very successful in data and communication intensive parallel applications such as multimedia data processing, scientific computing, and bioinformatics. However, to achieve the desired performance-power-reliability tradeoffs, suitable software support is critical for CMPs. In fact, it is clear that CMP hardware cannot evolve independently of the proper software infrastructure, and software development tools are really the key to realizing the benefits offered by CMPs. While an overwhelming majority of prior CMP software related efforts focused on performance or power optimizations, there are very few studies that target at improving hardware reliability. This is unfortunate, because as transistor sizes and voltages of electronic circuits continue to scale, one can expect reliability to be even more challenging for future CMPs.

There are several aspects of hardware reliability as far as CMPs are concerned. For example, correct execution of instructions is vital and can be helped with techniques such as dual execution [5]. Similarly, due to their relatively large sizes, memory components of a CMP can be vulnerable to hardware failures. Conventional methods for addressing potential memory related faults include error detection and correction codes. Another emerging problem area is the on-chip communication fabric. Since future technologies offer the promise of being able to integrate billions of transistors on a chip, the prospects of having hundreds of processors on a single chip along with an underlying memory hierarchy and an interconnection system will be entirely feasible. Once the number of cores on one CMP passes some threshold ($\sim$16 cores), conventional point-to-point buses will no longer be sufficient. These future CMPs will require an on-chip network (an NoC, Network-on-Chip [11]) in order to be able to handle the required communications between the cores in a scalable, flexible, programmable, and reliable fashion. Consider, as an example, a 2D mesh NoC that can be used to connect the cores in a CMP. There are several advantages of this kind of on-chip network. First, meshes work well with the conventional VLSI technology (which is 2D). Second, meshes scale very well as the number of cores is increased due their high bisection bandwidths and low diameter. Third, communication can be packet based and more regular, and as a result, it can easily be exposed to software for optimization purposes. Fourth, switches in a NoC can be used for strengthening signals flowing through them, helping to reduce data loses.

A NoC can be affected by both permanent failures (e.g., a link is broken) and transient errors such as crosstalks and coupling noises. Technology scaling makes this reliability problem even worse, demanding solutions in both hardware and software. However, since many CMP systems that require fault tolerance also work under severe power-performance constraints, any reliability optimization should be carried out considering the impact on power and performance. A daunting challenge, therefore, involves developing solutions for addressing the NoC reliability problems that are both performance and power aware. This is the problem addressed in this work.

* This research is supported in part by NSF grants CNS #0720645, CCF #0811687, CCF #0702519, CNS #0202007 CNS #0509251, by a grant from Microsoft Corporation, and by a Marie Curie International Reintegration Grant within the 7th European Community Framework Programme.

† The author is with the ANL now.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*LCTES'10,* April 13–15, 2010, Stockholm, Sweden.

Copyright © 2010 ACM 978-1-60558-953-4/10/04. . . $10.00

Our belief is that significant power-performance-reliability gains can be achieved by *exposing the NoC-based CMP architecture to the compiler* and letting the compiler optimize a given application code for both thread assignment and inter-thread communications. In this work, we present and evaluate a compiler directed, power-performance aware message reliability enhancement scheme for NoC-based CMPs. The proposed scheme improves on-chip communication reliability by duplicating messages traveling across the CMP nodes such that, for each original message, its duplicate uses a different set of links as much as possible (to satisfy performance constraint). In addition, our approach tries to reuse communication links across the different phases of the program to maximize link shutdown opportunities for the NoC (to satisfy power constraint).

Prior work studied performance-oriented compiler techniques for CMPs [10, 25, 31], application/IP block mapping schemes [4, 17], and link power optimizations [8, 9, 21, 26, 27, 34]. Our work is different from these studies in that it is oriented toward improving "reliability" under performance and energy constraints. There also exist several efforts that target modeling/improving network reliability [6, 30, 42, 44, 45]. To our knowledge, *this paper presents the first compiler-based approach to NoC reliability, or even to network reliability in general.*

### 1.1 Contributions

• We propose a compiler directed NoC reliability enhancement scheme that duplicates communication packets using non-intersecting paths. A unique characteristic of this scheme is that it is both performance and power conscious. In this approach, the compiler identifies program phases and solves the problem for each phase using integer linear programming (ILP). The solution times experienced in our experiments were not very high (between 16.1 seconds and 2.7 minutes on a 2GHz Sun Solaris machine). We also discuss performance oriented and power oriented variants of our baseline implementation.

• We present an experimental evaluation of the proposed scheme and compare its behavior to a hardware based reliability scheme. The results obtained using the parallelized versions of the SPECFP2000 benchmarks [35] clearly show that our approach is much more effective than alternate approaches to NoC reliability. We also observed that most of the time (more than 90%) our approach was able to send the original message and its replica over non-intersecting paths.

• To show that our approach can also be used along with profiling to handle a larger set of application codes, we also report results from three applications (mpeg, g.721, and specjbb) where the parallel code structure cannot be fully captured at compile time. In our experiments, we also compare our approach to two pure hardware based fault tolerant routing schemes. Our results indicate that the proposed scheme is better than these hardware based fault tolerant routing schemes in terms of performance, power, and reliability.

### 1.2 Roadmap

The rest of this paper is organized as follows. The next section introduces our NoC based CMP architecture. Section 3 introduces the main data structure (Unweighted Memory Access Graph, UMAG) used by our approach, and Section 4 explains how we identify phases in a parallel program using the UMAG. Our approach to reliability enhancement is described in Section 5, and the ILP formulations and an example are presented. Section 6 presents an experimental evaluation of our scheme, and Section 7 concludes the paper.

## 2. NoC Based CMP Abstraction

We focus on an NoC based CMP architecture where the nodes form a two-dimensional (2D) mesh. In this architecture, each node has a processor core, a memory, and a network interface. The specific NoC we focus on is a 2D mesh but our approach can be adapted to work with other NoC topologies, as long as the topology is exposed to the compiler. The nodes of this mesh are connected to each other using switches and bi-directional links. The on-chip memory in a node is organized as a hierarchy with each node having a private L1 cache and a portion of the shared on-chip memory space (i.e., this is a shared memory CMP). The latency of a data access in this shared on-chip space is a function of the distance between the requester core and the node that holds the data (similar to the NUCA concept [16, 22] except that our on-chip memory space is managed by compiler). We assume static thread and data mappings, i.e., before our approach is applied, parallel threads are mapped to the CMP nodes and data blocks are mapped to the on-chip memory spaces (we will discuss these mappings later in more detail). When a core requires a data element, it accesses that data from either the on-chip memory space of one of the CMP nodes or the off-chip memory space, depending on the location of the data. While we do not consider data migration/replication within the CMP in our baseline implementation, our approach can also be made to operate under an on-chip memory management scheme that employs data migration and/or replication.

In our discussion we use the terms "message" and "packet" interchangeably, though in reality a message is composed of multiple packets. Each packet in turn is composed of multiple flits. We further assume that all flits of a given message follow the same path on the NoC. Also, while the traditional reliability/fault-tolerance theory distinguishes between the terms "error," "fault," and "failure," in this work we use these terms interchangeably as long as the context/meaning is clear.

In our approach, the selection of the routing paths is done by the compiler. The hardware needed for such "compiler-directed routing" is similar to that used in the Intel Teraflops Processor [15] and [26]. We assume an NoC switch that supports two types of routings: default X-Y routing and compiler-directed routing. The former is used by some of the schemes against which we compare our approach. The header of each packet contains a flag bit indicating which routing mechanism is used for this packet. A packet using the default X-Y routing contains the id of the destination node in its header, as shown in the upper part of Figure 1. When a switch receives such a packet, it uses the X-Y routing algorithm. For a 5 $\times$ 5 network, the header of a packet that employs compiler-directed routing contains three fields (see the lower part of Figure 1): the hop counter (4 bits), the orientation (2 bits), and the routing command sequence (13 bits). For each switch on the path from source node to destination node, there is a corresponding bit in the routing command sequence of the packet, which (along with the orientation value) tells the switch which output port to use. Figure 3 gives the meaning of the routing commands for the different values of the orientation field. The hop counter is reduced by one each time the packet is forwarded from one switch to another. It becomes zero, when the packet has arrived at its destination node.

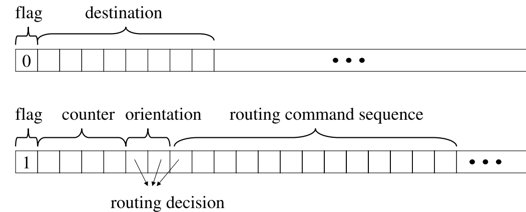


**Figure 1.** The fields in the header of a packet (Top: default X-Y routing; Bottom: compiler-directed routing).

| Orientation | 00 | 00 | 01 | 01 | 10 | 10 | 11 | 11 |
|---|---|---|---|---|---|---|---|---|
| Routing Command | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| Output Port | N | E | N | W | S | E | S | W |

**Figure 2.** Output ports used based on the orientation and routing command bits (N: North; S: South; W: West; E: East).

Future CMPs will contain a number of dynamic power optimizations. For example, each node or a set of nodes may be placed in a voltage island to support dynamic frequency and voltage scaling. The power feature that we are taking advantage of in this paper is NoC link shutdown [34]. When a link has been idle for some period of time, it is shut down to save energy. Shutting some of the links down may result in higher link sharing. This in turn can affect the performance of the application. In addition to this, link state transition (i.e., shutdowns and startups) overheads must be taken into account.

Note that multiple applications can be mapped to and executed on this CMP architecture concurrently. However, we assume that each application is assigned/given a contiguous partition of nodes of the CMP, and no communication links or cores are shared at the same time by the threads that belong to different applications. We

assume that the interface between the application and the OS allows the former to request a partition (a sub-mesh) from the latter.

## 3. Unweighted Memory Access Graph

Our approach has four steps. In the first step, we build a data structure that represents the parallel application and, in the second step, we use this structure to identify different program phases. The third step optimizes each phase in isolation for reliability enhancements using ILP. The last phase modifies the code to insert duplicate messages. Due to space limitation, we only discuss the details of the first three steps.

In the first step, the compiler analyzes the input code and builds a *Unweighted Memory Access Graph* (UMAG) to capture the memory access behavior of the parallel application based on the architecture defined in Section 2. A UMAG[1], which is built using static analysis, is a directed graph where each vertex represents a memory-related activity or a construct in the parallel code and each edge represents the flow. In mathematical terms, we map the given parallel program, $T$, to its graph representation $G(T)$, where

$$\begin{aligned} G(T) &= V(T) \cup E(T) \\ E(T) &\subseteq V(T) \times V(T). \end{aligned}$$

There are five types of vertices in a UMAG:

$$V(T) = L(T) \cup B(T) \cup A(T) \cup D(T) \cup W(T),$$

where a vertex $l \in L(T)$ represents a loop, more specifically, the entry point of a loop. Similarly, $b \in B(T)$ represents a back-jump of a loop, $ap \in A(T)$ represents an address packet transmission, $dp \in D(T)$ represents a data packet transmission, and $wp \in W(T)$ represents a write packet transmission. The $ap$ and $dp$ vertices capture the memory read activity, while the $wp$ vertices capture the memory write activity.

Data accesses are captured using vertices in $A(T)$, $D(T)$, and $W(T)$. More specifically, a data request is represented by a vertex $ap_i \in A(T)$; the actual data transfer is represented by a vertex $dp_j \in D(T)$. In the case of a write packet, both the address and the data to be updated are sent in one packet by $wp_k \in W(T)$. An edge $e \in E(T)$ is categorized based on the classification of the vertices it connects. We have:

$$E(T) = E_{Control}(T) \cup E_{Data}(T) \cup E_{Comp}(T).$$

There are three types of edges, namely, control edges ($E_{Control}$), memory access edges ($E_{Data}$), and computation edges ($E_{Comp}$).

• $E_{Control}(T) \subseteq B(T) \times L(T)$: A back-jump edge that connects a back-jump vertex, $b \in B(T)$, to a loop vertex, $l \in L(T)$.

• $E_{Data}(T) \subseteq D(T) \times A(T)$: A data access edge that connects a data packet vertex, $dp \in D(T)$, to an address packet vertex, $ap \in A(T)$, that belongs to a different loop nest.

Finally, $E_{Comp}(T) = E_L(T) \cup E_B(T) \cup E_A(T) \cup E_D(T) \cup E_W(T)$ gives the computation edges:

• $E_L(T) \subseteq L(T) \times (A(T) \cup D(T) \cup W(T))$: A control edge that connects a loop vertex, $l \in L(T)$, to either an address packet vertex, $ap \in A(T)$, or a data packet vertex, $dp \in D(T)$, or a write packet vertex, $wp \in W(T)$.

• $E_B(T) \subseteq (A(T) \cup D(T) \cup W(T)) \times B(T)$: A control edge that connects an address packet vertex, $ap \in A(T)$, or a data packet vertex, $dp \in D(T)$, or a write packet vertex, $wp \in W(T)$, to a back-jump vertex, $b \in B(T)$.

• $E_A(T) \subseteq A(T) \times (D(T) \cup W(T))$: A computation edge that connects an address packet vertex, $ap \in A(T)$, to a data packet vertex, $dp \in D(T)$, or a write packet vertex, $wp \in W(T)$.

• $E_D(T) \subseteq D(T) \times (A(T) \cup W(T))$: A computation edge that connects a data packet vertex, $dp \in D(T)$, to an address packet vertex, $ap \in A(T)$, or a write packet vertex, $wp \in W(T)$.

• $E_W(T) \subseteq W(T) \times (A(T) \cup D(T))$: A computation edge that connects a write packet vertex, $wp \in W(T)$, to an address packet vertex, $ap \in A(T)$, or a data packet vertex, $dp \in D(T)$.

[1] While not used in this paper, one can also envision a weighted version of the memory access graph (WMAG) for implementing link voltage scaling.

```
//Process 1
l1: for(...) {
        //request to read d1
        send_ap(1, 2, d1,...);
        //address packet ap1 is sent
        recv_dp(1, 2, d1,...);
        //data packet dp1 is received
        read(1, 2, d1,...);
        //reading d1 is finalized

        //compute d3 and send
        recv_ap(1, 2, d3,...);
        //address packet ap3 is received
        computing d3;
        //d3 is ready
        send_dp(1, 2, d3,...);
        //data packet dp3 is sent
}

//Process 3
l4: for(...) {
        //request to read d4
        send_ap(3, 2, d4,...);
        //address packet ap4 is sent
        recv_dp(3, 2, d4,...);
        //data packet dp4 is received
        read(3, 2, d4,...);
        //reading d4 is finalized

        //compute d2 and send
        recv_ap(3, 2, d2,...);
        //address packet ap2 is received
        computing d2;
        //d2 is ready
        send_dp(3, 2, d2,...);
        //data packet dp2 is sent
}
```

```
//Process 2
l2: for(...) {
        //compute d1 and send
        recv_ap(2, 1, d1,...);
        //address packet ap1 is received
        computing d1;
        //d1 is ready
        send_dp(2, 1, d1,...);
        //data packet dp1 is sent
        l3: for(...) {
                //compute d4 and send
                recv_ap(2, 3, d4,...);
                //address packet ap4 is received
                computing d4;
                //d4 is ready
                send_dp(2, 3, d4,...);
                //data packet dp4 is sent

                //request to read d2
                send_ap(2, 3, d2,...);
                //address packet ap2 is sent
                recv_dp(2, 3, d2,...);
                //data packet dp2 is received
                read(2, 3, d2,...);
                //reading d2 is finalized
                computing;
        }
        //request to read d3
        send_ap(2, 1, d3,...);
        //address packet ap3 is sent
        recv_dp(2, 1, d3,...);
        //data packet dp3 is received
        read(2, 1, d3,...);
        //reading d3 is finalized
}
```

**Figure 3.** Program code of a shared memory parallel program.

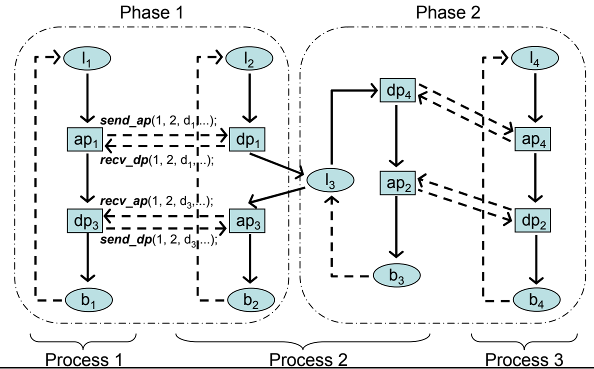


**Figure 4.** An example UMAG and the corresponding parallel computation phases.

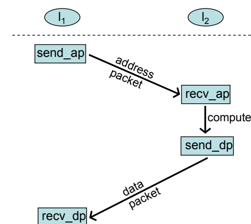


**Figure 5.** Details of an address/data packet transmission.

Figure 3 shows an example parallel code fragment composed of three different processes and four different loop nests ($l_1$ through $l_4$). Details of the computation statements as well as local memory accesses are not shown for clarity. However, read and write accesses to non-local memories are shown explicitly. Although there are many parameters involved in a read/write request, we are mostly interested in the source/target process (node) and the data element being accessed. More specifically, a read request is given as **read** (**$requester_{id}$, $process_{id}$, $data_{id}$,...**), where $requester_{id}$ is the requesting process (i.e., the one executing the read request), $process_{id}$ represents the process that is executing in the node that contains the memory which holds or will hold the requested data, and $data_{id}$ is the requested data item from that memory. Similarly, **write** (**$process_{id}$, $data_{id}$,...**) writes $data_{id}$ to the memory of the node that is executing $process_{id}$. In case of a write, a single write packet is sufficient since the address and the data to write can be transferred simultaneously in one packet.

Read and write accesses to non-local memories described above are represented with packets in our UMAG representation. Figure 4 shows the UMAG for the example in Figure 3. A read access is denoted by two packets, address and data. First, requesting process sends an address packet containing the address of the data element and the corresponding data packet contains the requested data element. However, as mentioned earlier, in case of a write, a single write packet is sufficient. This packet includes both the address and the value to be written. Note that data dependencies are also expressed using edges in our UMAG representation. Also, we use dashed edges to represent inter-iteration dependencies, whereas solid edges are used to represent intra-iteration dependencies. Lastly, the nodes used for data and address packets ($ap_i$, $dp_i$, and $wp_i$) do *not* mean that the actual packet transmissions occur at these nodes, rather they indicate that specific packets have been sent/received. Figure 5 shows the actual address and data packet transmissions in our UMAG representation. Edges represent the actual packet transmissions which, for clarity, we do not show in this much detail in Figure 4. A read request is an address packet followed by a data packet transmission. For example, as can be seen in Figure 3, process 1 initiates the the read request by sending an address packet, $send_ap(1, 2, d_1, ...)$. This request is received by process 2 with the $recv_ap(2, 1, d_1, ...)$ statement and the requested data is sent once the data is available (possibly after a computation step), through a data packet with $send_dp(2, 1, d_1, ...)$. This data packet is received by the requester with a $recv_dp(1, 2, d_1, ...)$ call. Although not shown here explicitly, a write request can also be represented similarly.

## 4. Phase Identification

After obtaining the UMAG representation, we divide it into *parallel computation phases* (PCPs). In this context, a PCP represents a set of loops that will execute in parallel (at runtime) and communicate with each other (through accessing some common data elements). We then apply our reliability enhancement scheme at a PCP granularity.

In order to formally express a PCP, we first define the concept of *loop connectivity*. Two loops are said to be connected if there is a packet transmission between them. For example, in Figure 4, loops $l_1$ and $l_2$ are connected due to data and address packet transmissions, whereas loops $l_2$ and $l_3$ do not have any connection. We express loop connectivity using $l_i \Longrightarrow l_j$ and formally define it as:

$$l_i \Longrightarrow l_j \quad \text{if } \exists e \in (E_A(T) \cup E_D(T) \cup E_W(T)) \text{ s.t. } e.source \in l_i,\ e.dest \in l_j,\ i \neq j.$$

Loop connectivity is a transitive property, that is, if $l_i \Longrightarrow l_j$ and $l_j \Longrightarrow l_k$, then we have $l_i \Longrightarrow l_k$. Similarly, we use $l_i \circ l_j$ to express that loops $l_i$ and $l_j$ are not connected. Using this notion of loop connectivity, collection of PCPs (or simply phases) form the loop nodes given with $L(T)$:

$$L(T) = \bigcup_{i=1}^{n} PCP_i, \text{s.t.} \quad l_j \circ l_k, \ \ \forall l_j \notin PCP_i \text{ and } \forall l_k \in PCP_i.$$

This definition implies that there cannot be a single $l_j \in PCP_s$ connected to another $l_k \in PCP_t$ directly or indirectly. For example, the UMAG in Figure 4 can be divided into two PCPs: $PCP_1 = (l_1, l_2)$ and $PCP_2 = (l_3, l_4)$.

Once the parallel program is decomposed into phases, the rest of our approach operates on one phase at a time. Since PCPs do not have any data dependencies between them, we can treat each phase as an independent execution unit and optimize it in isolation. For example, in Figure 4, there is no data dependence indicated by data/address packet transmissions between $PCP_1$ and $PCP_2$. This, in turn, allows our scheme to optimize two PCPs in isolation. Note that, the edges between $dp_1$ and $l_3$ as well as $l_3$ and $ap_3$ are control edges which do not prevent us from optimizing PCPs individually. Since each phase typically has a different inter-node communication pattern, it makes sense to formulate and solve a separate linear problem for each phase. Section 5 explains the ILP formulation we implement for a given phase (PCP).

## 5. Reliability Enhancement

In this section, we present our ILP formulation for the NoC reliability enhancement scheme that duplicates messages using non-intersecting paths. We implemented our ILP approach such that the number of links traversed between the source and the destination is minimum. Specific constraints that satisfy this property have not been shown explicitly for clarity reasons. We used Xpress-MP [43], a commercial tool, to formulate and solve our ILP problem. Note that, as explained above, the paths selected by the compiler may be different from those that would be adopted by conventional X-Y routing.

### 5.1 Reliability Centric Formulation

As explained earlier, we focus on a 2D mesh-based CMP, which is represented by a directed graph $G = (V, E)$. Each node in $V$ is assigned an identifier, e.g., $i$, and each edge in $E$ is denoted using its corresponding nodes, e.g., $(i, j)$. Using static program analysis as explained above, we first identify concurrent messages within each program phase. For example, if phase $n$ has $K$ concurrent messages to be transmitted, each message is represented by $(s_k, t_k, b_k)$ for $k = 1, \ldots, K$, where $s_k$ and $t_k$ are the source node and destination node, respectively, and $b_k$ captures the bandwidth required by this message.

We also employ a binary variable $M^{p,k}_{i,j}$ to describe the routing decision for message $k$ in phase $p$ at link $(i, j)$. Setting the value of this variable to 1 means that message $k$ is transmitted through link $(i, j)$; otherwise, message $k$ does not pass through link $(i, j)$. In order to capture the participation of node $n$ at the transmission of a message $k$ within phase $p$, we use $A_{p,k,n}$. More specifically, a node participates at the communication if a neighbor is part of the same communication and the link connecting these two nodes is active (not shutdown). We use a different 0-1 variable to capture the activity of a link:

$$E_{p,i,j} = \begin{cases} 1, & \text{if the link between } i \text{ and } j \text{ is active during phase } p. \\ 0, & \text{otherwise.} \end{cases}$$

Recall that, in our NoC-based CMP architecture, there are bi-directional links between neighboring nodes denoted by $i, j$. A link can be active in one phase of the program and inactive during the next phase. This enables us to control the NoC state, taking into account the communication requirements exhibited by a given phase. As explained earlier, to include a node in a communication activity, we need to ensure that a neighbor is part of the communication and the connecting link is active. This can be expressed as follows:

$$A_{p,k,j} \geq A_{p,k,i} + M^{p,k}_{i,j} - 1, \forall p, i, j. \tag{1}$$

In order to improve the reliability of the NoC, our approach duplicates the messages in the system. To capture this within our ILP formulation, we use $R^{p,k}_{i,j}$. Similar to $M^{p,k}_{i,j}$, this captures the communication behavior of the duplicate of message $k$.

$$R^{p,k}_{i,j} = \begin{cases} 1, & \text{if message } k \text{ is transmitted through link } (i, j) \\ & \text{during phase } p. \\ 0, & \text{otherwise.} \end{cases}$$

Like a regular message, a node will be included in the communication activity, if a duplicate message is being transmitted through:

$$A_{p,k,j} \geq A_{p,k,i} + R^{p,k}_{i,j} - 1, \forall p, i, j. \quad (2)$$

The neighbors indicated here correspond to the nodes on the north, south, west and east of the node represented by $A_{p,k,i}$. The nodes that are on the borders of the NoC have a subset of these constraints (depending on their specific locations). Note that, the source and the destination of each message are already known and given by $s_k$ and $t_k$, respectively. For each node designated as the source or the target, we set these variables to 1. More specifically, we have:

$$A_{p,k,i} = 1, \text{ if } i = s_k \text{ or } i = t_k, \forall p, k, i. \quad (3)$$

We need to ensure that, if a message is sent through a link, that link should be active. We capture this as follows:

$$E_{p,i,j} \geq M^{p,k}_{i,j}, \forall p, k, i, j. \quad (4)$$

Similarly, a link needs to be active during the transmission of a duplicate message over it:

$$E_{p,i,j} \geq R^{p,k}_{i,j}, \forall p, k, i, j. \quad (5)$$

We also need to make sure that both the original and the duplicate messages should follow different routes (paths) to the target. We can express this as follows:

$$M^{p,k}_{i,j} + R^{p,k}_{i,j} \leq 1, \forall p, k, i, j. \quad (6)$$

Next, we introduce $S_{p,i,j}$, the binary variable to indicate whether a link is shared by multiple messages during a phase:

$$S_{p,i,j} = \begin{cases} 1, & \text{if link } (i,j) \text{ is shared during phase } p. \\ 0, & \text{otherwise.} \end{cases}$$

To capture the behavior of this variable correctly, we need to consider all the message pairs including the original messages and duplicates. Consequently, we have:

$$\begin{aligned} S_{p,i,j} \geq & \quad M^{p,k_1}_{i,j} + M^{p,k_2}_{i,j} - 1, \\ S_{p,i,j} \geq & \quad M^{p,k_1}_{i,j} + R^{p,k_2}_{i,j} - 1, \\ S_{p,i,j} \geq & \quad R^{p,k_1}_{i,j} + R^{p,k_2}_{i,j} - 1, \\ & \forall p, i, j, k_1, k_2 \text{ such that } k_1 \neq k_2. \end{aligned} \quad (7)$$

If any two messages are identified to exercise a link, that link is marked as a shared link, that is, $S_{p,i,j}$ is set to 1.

We also need to capture the link state transitions (i.e., shutdowns and startups). It might be possible to hide the performance overhead due to these activations/deactivations by using a *preactivation* strategy (i.e., by activating a link ahead of the time before it is really needed so that it will be ready when it is needed). However, the energy overheads due to such activities cannot be hidden. To capture this overhead, we use $AE_{p,i,j}$ and $DE_{p,i,j}$ for activation (startup) and deactivation (shutdown), respectively. These constraints can be expressed as follows:

$$\begin{aligned} AE_{p,i,j} \geq & \quad E_{p,i,j} - E_{p-1,i,j}, \\ DE_{p,i,j} \geq & \quad E_{p-1,i,j} - E_{p,i,j}, \ \forall p, i, j. \end{aligned} \quad (8)$$

In the above expression, we check each communication link's activity in neighboring phases (PCPs) $p$ and $p-1$. If there is any change in the activity (state) of any link (i.e., any transition), one of the corresponding variables ($AE$ or $DE$) will be triggered (i.e., the corresponding variable will be set to 1). This overhead is included in our objective function.

Having specified the necessary constraints to be satisfied by our ILP formulation, we next discuss our objective function for reliability enhancement. We define our cost function (to minimize) as the sum of two separate cost factors: one to capture the performance concern and the other to capture the energy concern. Thus, our objective function can be expressed as follows:

$$\min \quad \sum_{p=1}^{P} \sum_{i=1}^{n} \sum_{j=1}^{n} C_1 \times S_{p,i,j} + C_2 \times (AE_{p,i,j} - DE_{p,i,j}). \quad (9)$$

In this expression, $C_1$ and $C_2$ capture the weights for the performance concern and energy concern, respectively. Note that, $S_{p,i,j}$ is used for the number of shared links, whereas $(AE_{p,i,j} - DE_{p,i,j})$ represents the number of links activated during this phase. The value of $(AE_{p,i,j} - DE_{p,i,j})$ could be negative meaning that the specific link is deactivated at the given phase. Also, if $C_1 > C_2$ the solution found will be more oriented towards improving performance (by minimizing the number of links shared by the messages in a given phase), whereas $C_1 < C_2$ favors an energy oriented solution (by maximizing the link reuse between neighboring phases).

### 5.2 Performance Centric Formulation

The formulation presented above duplicates every original message in the phase. This can have performance consequences despite the fact that our approach tries to route original and duplicate messages using non-intersecting paths as much as possible. In this subsection, we present an alternate formulation which favors performance. Specifically, we try to maximize the number of duplicates while not allowing any links to be shared by two or more messages (duplicate or original) in a given phase. We have to make several modifications to the reliability centric formulation presented above to obtain this performance centric formulation. First, Expression (7) above should be modified in order to capture this new constraint. If we consider two different messages, $k_1$ and $k_2$, we will have four variables, $M^{p,k_1}_{i,j}$ and $M^{p,k_2}_{i,j}$ for the original messages, and $R^{p,k_1}_{i,j}$ and $R^{p,k_2}_{i,j}$ for their duplicates. All these messages should follow different routes in order satisfy the minimum performance overhead constraint. This can be captured as follows:

$$\begin{aligned} & M^{p,k_1}_{i,j} + M^{p,k_2}_{i,j} + R^{p,k_1}_{i,j} + R^{p,k_2}_{i,j} \leq 1, \\ & \forall p, i, j, k_1, k_2 \text{ such that } k_1 \neq k_2. \end{aligned} \quad (10)$$

We also need to modify the constraints to reflect the fact that a duplicate may not exist. To do this, we introduce another binary variable, $RME_{p,k}$, which indicates whether there exists a duplicate for message $k$ in phase $p$. All the routing variables related to the duplicates are dependent on this variable. More specifically,

$$R^{p,k}_{i,j} \leq RME_{p,k}, \forall p, k, i, j. \quad (11)$$

Similarly, our objective function given originally by Expression (9) has to be modified to reflect this change:

$$\begin{aligned} \min \quad \sum_{p=1}^{P} \quad & (C_2 \times \sum_{i=1}^{n} \sum_{j=1}^{n} (AE_{p,i,j} - DE_{p,i,j})) + \\ & (C_3 \times \sum_{k=1}^{K} (1 - RME_{p,k})). \end{aligned} \quad (12)$$

This objective function tries to minimize the energy consumption and maximize the number of duplicate messages. The portion preceded by a weight of $C_2$ captures the energy metric, whereas the portion preceded by a weight of $C_3$ captures the reliability metric. $RME_{p,k}$ indicates whether the duplicate exists and $\sum_{k=1}^{K}(1 - RME_{p,k})$ sums up the non-duplicated messages.

### 5.3 Energy Centric Formulation

Similar to the performance centric routing, we may formulate an energy centric routing as well. In this case, duplicate messages are not routed through links that were not active in the previous phase, that is, the extra energy consumed due to reliability enhancement is reduced. Expression (5) above ensures that a duplicate message is transmitted through link if that link is already active. In addition to this, our goal is not to keep a link active if only duplicates are transmitted through this link:

$$E_{p,i,j} \leq \sum_{k=1}^{K} M^{p,k}_{i,j}, \forall p, k, i, j. \quad (13)$$

The right-hand side of the above constraint captures the total number of original messages transmitted over the link. If this sum is 0, then the corresponding link will not be active, forcing duplicates to follow different routes. In addition to this, we introduce $RME_{p,k}$ to indicate

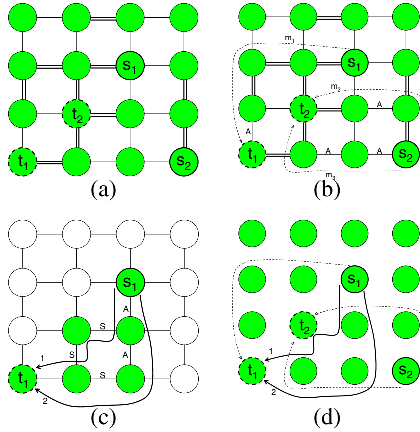


**Figure 6.** An example showing our ILP-based approach.

whether a duplicate of message $k$ exists during phase $p$. This is similar to $RME_{p,k}$ that is used in Section 5.2. Hence, we use the corresponding constraint given in Expression (11) to indicate that a duplicate message can be transmitted only if the duplicate exists.

Our new objective function can be written as:

$$\min \quad \sum_{p=1}^{P}(C_1 \times \sum_{i=1}^{n}\sum_{j=1}^{n} S_{p,i,j}) + \quad (14)$$

$$(C_2 \times \sum_{i=1}^{n}\sum_{j=1}^{n}(AE_{p,i,j} - DE_{p,i,j})) + (C_3 \times \sum_{k=1}^{K}(1 - RME_{p,k})).$$

Coefficients $C_1$, $C_2$, and $C_3$ can be selected based on the relative weights of the performance concern, energy concern, and reliability concern, respectively. We have all three weights ($C_1$, $C_2$, and $C_3$), since we still need to capture the energy concern. This follows from the fact that we still have link activations and deactivations.

### 5.4 Example

Figure 6 shows an example $4 \times 4$ NoC and the corresponding ILP solution based on our baseline formulation discussed in Section 5.1. In this example, we assume that there are two original messages that have to be transmitted in a phase: a message from $s_1$ to $t_1$ and a message from $s_2$ to $t_2$. Source nodes are represented using solid circles, whereas the destination nodes are represented using dashed circles. Furthermore, we assume that active links from the previous phase are known and denoted using double edges between the NoC nodes. Figure 6(a) shows the messages, their source and destination nodes, and the active links.

With our baseline (reliability centric) formulation (i.e., all messages have their attached duplicates which follow different paths from the original messages as much as possible), three of the four messages (two original and two duplicate) will follow the routes given in Figure 6(b). Note that the shortest paths are used in order not to increase energy consumption. Similarly, for $m_1$, the ILP solver returns this route as one of the routes for messages from $s_1$ to $t_1$ no matter what the $C_1$ and $C_2$ parameters are. In order to satisfy these routes, some links that were not active in the previous phase will need to be activated. These links are identified by attaching an $A$ next to the edges in Figure 6(b). In total, there are 4 additional link activations required to transmit the 3 messages in this phase.

So, the question becomes how to route the duplicate message from $s_1$ to $t_1$. There are two different routes, marked using 1 and 2 in Figure 6(c). If this message is sent through route 1, there will be an additional activation required on the first link. On top of this link activation, one of the links will be shared with $m_2$ and another link will be shared with $m_3$. Instead, if this message is sent through route 2, there will be two link activations followed by a shared link with $m_3$. Figure 6(d) shows the overall behavior of the system during this phase. Either we will follow route 1 and incur $(4A) + (2S + A) = 2S + 5A$, or we will follow route 2 and incur $(4A) + (S + 2A) = S + 6A$. If we consider our objective function in Expression (9), the cost function will be $2 \times C_1 + 5 \times C_2$ for route 1, and $C_1 + 6 \times C_2$ for route 2. At this point, selecting a route from these two depends on the values of $C_1$ and $C_2$. If $C_1 = C_2$ either one of the routes could be chosen by the ILP solver. However, if $C_1 > C_2$, the objective function will be minimized when route 2 is selected. This follows from the fact that weight of performance is now increased, i.e., the route with fewer shared links would be preferred (as in the case of route 2) over the alternate route with lower energy consumption. By comparison, if $C_1 < C_2$, route 1 will be chosen since, in this case, link sharing is preferred over link activation.

**Table 1.** Our simulation parameters and their default values.

| Parameter | Value |
|---|---|
| NoC | $5 \times 5$ 2D mesh |
| Core | two-issue |
| CPU Frequency | 1GHz |
| Data/Instr L1 Capacity | 8KB (per node) |
| Local On-Chip Memory | 256KB (per node, banked) |
| Link Speed | 1GHz |
| Link Activation Latency | 1 $\mu$sec |
| Link Activation Energy | 140 $\mu$joule |
| Packet Header Size | 3 flits |
| Flit Size | 39 bits |

### 5.5 Qualitative Comparison Against Existing Schemes

In this section, we discuss how our compiler-directed message duplication approach compares to hardware level approaches for NoC reliability. There are several approaches in the literature that target improving the resilience of NoCs using error detection and correction mechanisms. Most error detection mechanisms target transient link errors and try to cope with them by attaching an error code to each packet to be sent over the network. This error code, typically a parity or cyclic redundancy check, is used to detect at the destination whether the content of the packet has been modified or not. Such schemes typically require retransmission of the packet if an error is detected in it (the error is signaled to the source node using a negative ACK (NACK) signal). An alternate option would be to use a more sophisticated (error protection) code to allow the target node itself to correct the error without requiring any retransmission. As pointed out by prior research [39], as far as power consumption and implementation complexity are concerned, the first option (detection followed by retransmission) is preferred over the second one (self-correction). Therefore, in this paper we consider only the first option, and when we refer to error protection code we mean one that can detect (using a parity bit) an odd-number of errors. In the rest of this paper, we use *PEC* (Parity/retransmission based Error Correction) to refer to this error protection code based approach.

Such protection codes can only be useful in the context of transient link errors. In case of permanent link failures (e.g., broken links), they will not work as the original packet would not have arrived at its destination at the first place. For permanent link failures, our approach is much more effective, as it sends two copies of the same packet over non-intersecting paths as much as possible.[2] From the perspective of transient errors, the comparison between our approach and error protection based schemes is more involved. This is because our scheme can also be used in conjunction with error protection coding, that is, both copies can be augmented with an error protection code. If the copies are not protected with encoding, there are two cases to consider. If both of the copies reach the destination and they are the same, chances of transient error(s) having occurred are very low. Because an error would have had to flip the same bits in both the copies to escape detection. On the other hand, if only one of the copies arrives at the destination (due to permanent errors), there is no way to detect a transient error. If, however, the copies are augmented with protection codes, at the destination we can check both copies and accept the one without errors (based on parity bits), or ask for a retransmit if only one copy arrives.

[2] Note that our scheme does not require "the knowledge of which links will fail" at compile time. Instead, it prepares for every scenario, by trying to send any message and its duplicate using non-intersecting paths.

To summarize, if permanent link failure occurs, our approach has a clear advantage over conventional error protection based schemes, irrespective of whether transient errors also occur or not. On the other hand, if only transient errors occur and both the copies arrive, our approach can potentially detect more errors than conventional parity or CRC based schemes, since we can perform bit-by-bit comparison of the original message and duplicate message. In our experiments, we evaluate the impact of our approach in mitigating the effects of both permanent and transient errors, and compare it with PEC.

There are also approaches in the literature that can be defined as "hardware based fault tolerant routing" [29, 32, 33, 42, 44]. In many prior studies, researchers propose using virtual channels. Most of these approaches employ algorithms that are based on seminal works such as [13] and [37]. However, if the physical links are broken, such schemes may not be very effective. Another option is to reserve a set of physical links to route a message that could not be routed due to the failure of one or more of the links in its original path. Note that, apart from the implementation complexities involving in detecting the failure and re-routing the message, this approach also reduces the effective network size. To compare our approaches to hardware based fault tolerant routing schemes, we implemented this physical link reservation based approach (referred to as HFT-1 here) and an alternate hardware based fault tolerance scheme based on link state sharing, which is denoted using HFT-2. While there exist several implementations of such link state sharing based schemes, the implementation we adopted is based on the work described in [1]. In this implementation, each router periodically updates its neighboring routers with its health (e.g., which, if any, of its links have failed). In the long run, it is expected that the NoC converges to a stable state where every router has an idea about the global NoC health. Note that the periodic updates can flood the on-chip network and also cause extra power consumption. Therefore, in the implementation of [1] (and in ours as well), instead of using periodic updates, we perform updates only when a link fails. The routers, after receiving the new state information, remove the faulty link from their routing tables and exchange this new information with their own neighbors, and so on. At periodic intervals, each router calculates the shortest paths using Dijkstra's algorithm. Our preliminary experiments showed that both HFT-1 and HFT-2 are more energy efficient than the schemes discussed in [33].

Before moving to our experiments, we want to mention that the paths selected by our scheme do not lead to a deadlock since we have an additional set of constraints in our ILP formulation to prevent potential deadlocks. For the sake of clarity, we do not present our deadlock prevention constraints in detail. A deadlock will happen when there is a circular message dependency between two or more messages. We first identify the deadlock-possible messages and form deadlock sets. We then use these deadlock sets to generate our additional deadlock prevention constraints. If, for example, two messages $M_{i,j}^{p,k_1}$ and $M_{j,i}^{p,k_2}$ can cause a deadlock, then we add an additional constraint, $M_{i,j}^{p,k_1} + M_{j,i}^{p,k_2} \leq 1$, to prevent both of them being enabled. We define these constraints for the replica messages as well.

# 6. Experiments

## 6.1 Implementation and Setup

We implemented our compiler directed approach using the SUIF infrastructure [14] and performed experiments with all the applications in the SPECFP2000 benchmark suite [35]. For each of the benchmarks in our suite, we fast-forwarded the first 500 million instructions and simulated the next 2 billion instructions. The default values of our simulation parameters used in our experiments are listed in Table 1.

Our approach is implemented as a separate compilation phase within SUIF. Once the SUIF based analysis is performed, the collected information is passed to our ILP solver (Xpress-MP [43]). The solutions returned by the solver are mapped to SUIF and are used to modify the code to insert duplicate messages and specify the paths (routes) that will be used by the original and duplicate messages. The code modifications required for specifying the routes are similar to those in [8]. The overall compilation times we experienced on a 2GHz Sun Solaris machine varied between 19.8 seconds and 2.9 minutes (dominated by the ILP solution times). These solution times, which correspond to about 30% increase over the original compilation times, are not very large since we formulate a separate linear program for each phase (PCP) in isolation. Also, the maximum increase in code size due to our scheme was less than 3%, and its impact on instruction cache performance was negligible.

**Table 2.** Benchmarks used in our experiments and their important characteristics. Energy values are in mJ, and the latency values are in million cycles.

| Benchmark Name | Number of Phases | Number of Messages | Execution Cycles | Energy Consumption |
|---|---|---|---|---|
| wupwise | 72 | 13.5M | 388.1M | 781.6mJ (19.3%) |
| swim | 87 | 22.8M | 477.3M | 886.1mJ (26.6%) |
| mgrid | 64 | 11.7M | 406.0M | 814.2mJ (22.8%) |
| applu | 59 | 26.9M | 461.6M | 759.7mJ (15.7%) |
| mesa | 51 | 14.1M | 318.2M | 582.8mJ (19.0%) |
| galgel | 89 | 16.9M | 386.7M | 609.8mJ (24.8%) |
| art | 37 | 13.2M | 297.3M | 424.9mJ (33.2%) |
| equake | 59 | 9.4M | 192.6M | 387.2mJ (20.1%) |
| facerec | 73 | 6.3M | 208.9M | 390.5mJ (27.4%) |
| ammp | 113 | 12.7M | 241.1M | 407.8mJ (24.4%) |
| lucas | 96 | 5.7M | 156.4M | 321.4mJ (19.7%) |
| fma3d | 91 | 9.6M | 197.4M | 456.3mJ (29.3%) |
| sixtrack | 76 | 8.2M | 148.5M | 292.3mJ (28.6%) |
| apsi | 137 | 12.7M | 276.8M | 416.7mJ (22.5%) |

Before our approach is applied, two other phases (steps) are executed: *code parallelization* and *thread-to-core mapping*. The code parallelization phase determines loop iteration distribution and data decomposition across the CMP nodes. The specific method used in this phase is very similar to the approach in [3], except that most frequently used data are mapped to the on-chip memory components. The second step applied before our approach is thread-to-core mapping (which we will discuss shortly). After these two steps, our approach is invoked. Note that, while we use specific code parallelization and thread mapping schemes in our experimental evaluation, our approach can work with other parallelization/mapping schemes as well. In our experiments, we also present results with different thread-to-core mappings. Also, in all our experiments, all processor cores are used but the set of links used depend on the communication pattern and message routing strategy.

To conduct our experiments, we implemented a flit-level network-on-chip simulator (built on top of Orion [41]) and connected it with SIMICS [40], a multi-processor simulator. The network is parametrized in a similar fashion to that in [11] except that it is $5 \times 5$. The link speed is set to 1Gb/sec. Each input port of switch has a buffer that can hold 64 flits, each of which is 128 bits wide (packet size is 16 flits). The communication links in this network can be shutdown independently, using a time-out based mechanism as described in [34]. We set the time-out counter threshold for the hardware-based power reduction scheme to 1.5$\mu$sec based on some preliminary analysis. The time it takes to switch a link from the power-down state to the active state is set as 1$\mu$sec, and the energy overhead of this switching is assumed to be 140$\mu J$ based on prior research. For modeling the energy consumption of memory components, we used CACTI [38], and for collecting energy data for core-related activities, we enhanced SIMICS with accurate timing models and energy models similar to those employed in [7]. The NoC energy modeling is based on Orion [41]. All the energy numbers presented include both dynamic and leakage energy.

Table 2 presents the important statistics for our benchmarks under the default values of our simulation parameters. The second column lists the number of phases (PCPs) for each benchmark and the next one shows the number of messages sent over the NoC during the entire simulation time. The fourth column gives the execution cycles and the fifth column shows the total energy consumption of the CMP when no reliability optimization is applied. These values include the energy spent in the datapath, on-chip and off-chip memory accesses, and interprocessor communication. The values within parentheses in the last column give the contribution of the NoC to the total energy consumption of the chip.

In our evaluation, for each application code in our experimental suite, we performed experiments with different versions (in addition to the *original version* that does not perform any reliability optimizations but tries to maximize link reuse between neighboring phases to reduce energy consumption); the details of these versions will be explained in Section 6.2. In evaluation of all the versions, including the original one, we assume that a power-saving scheme for NoC is

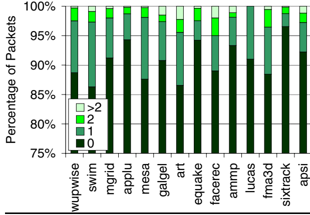


**Figure 7.** Link sharing statistics for packets.

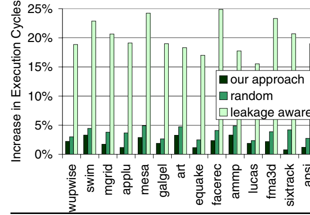


**Figure 8.** Performance degradation caused by different versions.

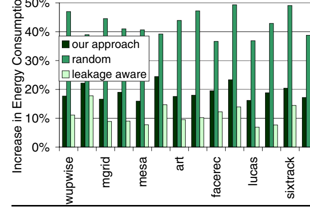


**Figure 9.** Additional energy consumption caused by different versions.

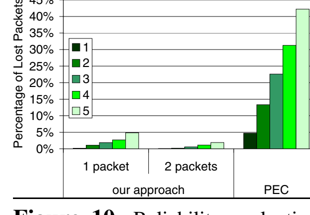


**Figure 10.** Reliability evaluation of our approach and PEC regarding permanent link failures.

already in place. Specifically, as stated earlier, we assume the existence of a hardware based link power saving scheme that turns off a link if it has not been used for a while. Note that the original scheme and optimized schemes use the same parallelization and thread mapping steps. Also, we model the overheads incurred by our approach in detail, and the performance and power numbers presented below include *all* overheads (e.g., cost of link shutdowns and startups and additional link contention due to duplicate messages).

### 6.2 Results

Unless otherwise stated, when we say "our approach" in our discussion below, we mean the reliability centric formulation given in Section 5.1. We start by presenting the increase in message traffic, energy consumption and execution cycles when our reliability centric formulation is used, assuming $C_1 = C_2$ in the objective function. All the numbers presented below are with respect to the corresponding values obtained under the original scheme, i.e., they are given as percentage degradation over the original scheme (see Table 2 for the absolute values with the original scheme). Figure 7 gives, for our scheme, the fraction of packets that share 0, 1, 2 and more than 2 links (denoted $> 2$), over all the phases of the application. We see that a large fraction of the packets do not share any links, that is, our approach is able to send the original and duplicate packets along the disjoint paths most of the time (90.1% on average). Of the cases when this is not possible (i.e., there is at least one link shared), the most frequent reason is that the source and destination nodes reside along the same row or column.[3] In some other cases, we simply could not find disjoint paths due to large number of packets that have to be routed. To discuss the performance degradation and energy increase caused by our approach, let us consider the first bars, for each application, in Figures 8 and 9.

Our main observation from Figure 8 is that the performance degradation caused by our approach – over the original version – is not high, and varies between 0.7% and 3.3%. Again, this is due to the success of our scheme in finding *non-intersecting* paths, as much as possible, for the duplicate packets. The increases in energy consumption – given in Figure 9 – are higher, mainly due to the increase in dynamic energy consumption as a result of duplicate messages. While our approach keeps the increase in leakage consumption at minimum by maximizing the link reuse between neighboring program phases, it still has to send extra packets (duplicates), which contribute to the dynamic energy consumption. As a result, on average, our approach incurs a 19.1% increase in total energy consumption. We also performed experiments with different $C_1$ and $C_2$ values. When we set $C_1 = 2$ and $C_2 = 1$, we observed that the average performance penalty reduced to less than 1%, but the average energy increase jumped to 26.6%. In contrast, when $C_1 = 1$ and $C_2 = 2$, the percentage increase in execution cycles and energy consumption became 3.8% and 9.7% (on average). These results show that by changing the values of $C_1$ and $C_2$, one can explore the tradeoffs between performance and energy loses, and select the appropriate reliability-centric solution that satisfies the performance and power constraints at hand. While, for a given power/performance bound, several search algorithms can be developed for determining the best $C_1$ and $C_2$ values, studying such algorithms is beyond the scope of this work. We have already implemented in our compiler a simple heuristic that selects the best $C_1$ and $C_2$ values for a given power/performance bound. We could not present here the details of this heuristic due to lack of space.

We now compare this baseline implementation of ours against two alternate schemes. The first scheme, called Random, selects the paths for duplicate messages randomly among all paths of minimum links (between the source and destination). Due to its random nature, we can expect this scheme to perform reasonably well as far as performance is concerned. As can be observed from the second bars in Figure 8, the average performance degradation this alternate scheme causes is about 3.7% (compared to 2.1% caused by our scheme). However, since this approach does not care about the link reuse between neighboring phases of the application, it can cause significant increase in leakage energy, as can be observed from Figure 9. It leads to an average increase of 42.6% on total energy consumption. The second alternate scheme we experiment with is fully leakage oriented. Specifically, the duplicate packets in this scheme are always routed along the same path as the corresponding original packets. While this approach is leakage efficient in general, it can also lead to a significant increase in execution latency. We see from Figures 8 and 9 that it increases execution latency (resp. energy consumption) by 20.1% (resp. 1.8%). Therefore, considering both performance and energy consumption, our performance-energy conscious reliability enhancement clearly strikes the right balance. Though not presented here, the average performance and energy degradations caused by the PEC scheme were 3.9% and 4.2%, respectively.

Our next set of experiments study the reliability improvement brought by our approach in more detail. For this purpose, we evaluated the behavior of our approach under both permanent and transient errors. In the case of permanent errors, we simulated the case when a certain number of links are disabled and recorded the fraction of packets that did not arrive at their destination nodes.[4] The results are presented in Figure 10, each bar representing the *average value* across all applications. For each application, we have two versions: 1) our approach without any error encoding and 2) error encoding (PEC) without our approach. We experimented with different numbers of link failures (from 1 to 5), and the y-axis in Figure 10 gives the fraction of packets that could not reach their destinations due to a broken link. In the case of our approach, we also quantified the fraction of packets when both copies failed, and those when only one copy failed. We see that, with our approach, only a small fraction of the packets could not reach their destinations. The reason for such low figures is because our approach is, in general, able to route the two copies using non-intersecting paths. By comparison, the safe delivery rate under the PEC scheme can be very low, indicating that, as far as permanent failures are concerned, our approach is very promising, especially when a large number of links are permanently disabled.

We also quantified the benefits of our approach when transient errors occur. To cover a large set of possibilities, we experimented with different error probabilities. We used the same two schemes above (our approach and PEC). Figure 11 shows the number of errors that could not be detected under each scheme. As before, each bar represents an average value across all benchmarks. Note that in PEC an error may not be detected if it affects an even number of bits. On the other hand, in our scheme, we may fail to detect an error if both copies have the same bit (or set of bits) flipped. Clearly, the likelihood of this is extremely low since the copies usually go over non-intersecting paths. The results in Figure 11 indicate that our approach performs much better than the PEC scheme in the case of transient errors, for all the error injection rates considered.

Let us now quantify the cost of correcting the detected errors in terms of execution cycles. In the case of PEC, to correct the

[3] As explained earlier, our baseline approach always uses the minimum number of links between the source and destination.

[4] The fraction of messages lost in transmission is a frequently-used reliability metric in fault-tolerant network research [44].

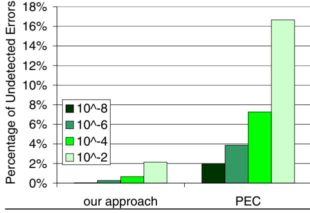


**Figure 11.** Reliability evaluation of our approach and PEC regarding transient link errors.

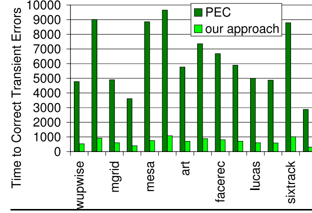


**Figure 12.** Time (in cycles) to correct the transient errors detected.

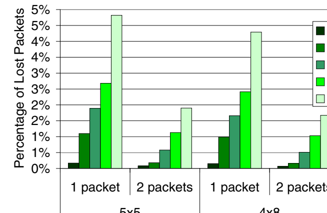


**Figure 13.** Impact of CMP size regarding permanent link failures.

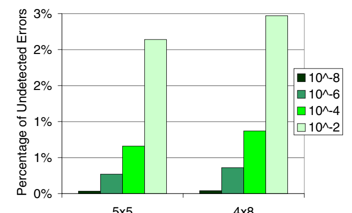


**Figure 14.** Impact of CMP size regarding transient link errors.

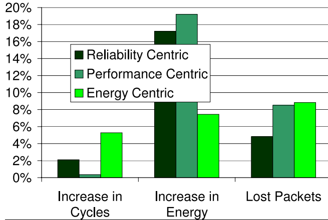


**Figure 15.** Comparison of reliability centric, performance centric and energy centric formulations.

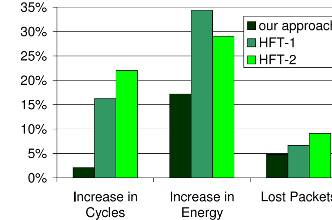


**Figure 16.** Comparison of our approach against two hardware based fault tolerant routing schemes.

error, the packet is retransmitted. Therefore, the total performance cost of error correction is the time required to retransmit the packet. For our approach, however, the actual cost depends on whether we employ any error protection code. If the copies are not protected with error codes, our approach cannot correct the errors it detects by message comparison (but we can always request a retransmission, if desired, as in the case of the PEC scheme). If, on the other hand, the copies are protected using an error protection code, we can easily select the copy without the error (if there is one, which is highly likely), and therefore, the error correction time is expected to be very short. Figure 12 gives, for the benchmarks in our experimental suite, the average time to correct the detected errors (note that the actual time to correct an error depends on the distance between the source and destination). The results with our approach assume that the packets are augmented with parity. We see that, as expected, the error corrections times with our approach are lower than the corresponding times with the PEC scheme.

As stated earlier, we assume that thread-to-core mapping has already been performed before our approach is applied. While the choice of the thread/data mapping scheme used is orthogonal to the focus of our approach, we may achieve power/performance values of different magnitudes, depending on the thread/data mapping used. In our default mapping scheme, we used an affinity-based approach, wherein the parallel threads are assigned to cores based on the data sharing between them. To do this, the compiler estimates the data sharing between each pair of threads, and the pairs that have high affinity (i.e., share a lot of data between them) are assigned to neighboring nodes of the CMP. This helps reduce the cost of interprocessor communication. Also, each data element is assigned to the on-chip memory of the node that uses that element most. The infrequently-used data elements are assigned to the off-chip memory. We changed our thread mapping scheme and performed a sensitivity analysis. The main difference between our new mapping and our default mapping is that the new one maps the parallel threads to CMP nodes randomly. However, the parallelization step used along with both the schemes is still the same. This allows us to see how our savings change when we do not exploit the affinity among parallel threads. We found that the percentage increases in execution cycles and energy consumption with this alternate thread mapping were very similar to those obtained with the default mapping used so far. Also, while both the percentage of lost packets and the percentage of undetected transient errors increased a little (less than 2.2%) with this alternate mapping (as a result of the increased number of links that have to be traversed by a packet), our approach still generated better reliability results than PEC.

Our next set of experiments study the sensitivity of our approach to network size. Recall that the default network size we have is $5 \times 5$. We also performed experiments with a larger network ($4 \times 8$), and the results are presented in Figure 13 (for permanent link failures) and Figure 14 (for transient errors). Our approach is more successful with permanent failures with larger networks. This is because a given broken link has less chance of affecting the packets traveling across the network. However, we also observe a reverse trend when we look at the transient errors. More specifically, as the network size is increased, the percentage of undetected errors go up slightly. While not presented here in detail, we also performed experiments with different types of cores (e.g., single issue versus four issue). The results observed were not very different compared to our default core configuration which is two issue. Similarly, when we reduce/increase our default switch buffer capacity (default was 64 flits), the results did not change too much (within 2%).

We now summarize the results of our experiments with performance centric and energy centric formulations discussed in Sections 5.2 and 5.3, respectively. The increase in execution cycles, increase in energy consumption and percentage of lost packets (when 5 links fail) are presented in Figure 15 for the reliability centric, performance centric and energy centric formulations. Each bar represents the average values across all applications we tested and is obtained when $C_1$, $C_2$, and $C_3$ (when used) are set to 1. One can see from these results the tradeoffs between performance, power and reliability. Depending on the constraints at hand, these three different formulations can be explored to reach an acceptable solution. We also want to emphasize that, as mentioned earlier, for each formulation we have also flexibility of changing the values of $C_1$, $C_2$, and $C_3$.

As stated earlier, we also implemented and performed experiments with two hardware based fault tolerant routing schemes: HFT-1 and HFT-2. Figure 16 presents the increase in execution cycles, increase in energy consumption, and fraction of lost packets (again, under the assumption of 5 link failures). In our HFT-1 implementation, we reserved 8 links (that cover an area from the upper left corner to the lower right corner) to be used when any other links fails. The failed links are selected randomly from among the remaining links. Each bar represents the average value when all our applications are considered. The main observation from these results is that our approach generates better results than these two hardware based schemes for all these three metrics. The reason that HFT-1 has high performance overhead is that, under this scheme, part of the network is not available for use as long as there are no failures. While this may not hurt performance much in large chip-to-chip networks, in relatively small NoCs such as ours it may be a big performance bottleneck. The reason that HFT-2 has high performance overhead is because of two factors. First, sending updates of routing tables to neighboring nodes floods the network with messages (this is in fact a general problem with many adaptive routing algorithms). Second, from time to time, the Dijkstra's shortest path algorithm is executed by the switches (in our implementation by the cores) to ensure that the routing tables are up-to-date. These two factors also contribute to the high power consumption. HFT-1 cannot completely eliminate lost packets if one (or more) of the failed links is directly connected to the reserved path and this link is the only connection between a node and the reserved path. Similarly, in the HFT-2 case, since it is not possible to keep all routing tables up-to-date all the time, it is not possible to eliminate all the lost packets. It is reasonable to expect that both HFT-1 and HFT-2 are much more costly to implement – as far as circuit complexities and area demands are considered – than our approach, so, we believe our approach is preferable over the hardware based fault tolerant schemes from the circuit angle as well.

Our approach is a static one meaning that the UMAG is determined at compile time using static analysis. However, in some cases, we may not have the complete information – at compile time – to

build the UMAG and identify the phases. In such cases, our approach can be used with *profile information.* More specifically, the application code can be profiled to collect inter-core communication information and this profile data can be used to build the UMAG. Note that, in this case, with a different input, the actual communications can be different from those captured by the UMAG; however, this does not create any correctness issue. In the worst case, we incur extra execution cycles and extra power consumption but the program semantics do not change. To check the feasibility of this approach, we first considered two applications from the MediaBench suite [24], mpeg and g.721, whose UMAGs cannot be completely and accurately captured using static compiler analysis alone. We profiled them, built their UMAGs, and applied our reliability enhancement scheme. Then, we executed the applications with different input sets to measure input sensitivity. We found that the average performance degradation with both the applications varied between 2.4% and 4.1% depending on the input, and the average increase in energy consumption ranged from 16.6% to 21.3%, again depending on the input used. We also observed that, as compared to the PEC scheme, our approach generated much better reliability results.

We also performed experiments with specjbb [36] which is a server application that can really put pressure on the NoC. We ran this benchmark with 12 warehouses, with one client per warehouse, and the UMAG of this application was extracted again using profiling. Due to lack of space, we do not provide the details of how the application code is modified to enable message routings. The experiments with our scheme showed that the increases in execution cycles and energy consumption were around 9.6% and 23.4%, respectively. Clearly, these values are higher than the average values observed with the SPEC benchmarks (and the main reason for this in the high link sharing across different messages). However, when we move to our performance centric formulation, the increases in cycles and energy moved to 3.8% and 26.1%, with only a small (2.1%) increase in the fraction of lost packets. Therefore, we believe that, depending on the target optimization metric, our approach can be used for improving reliability for this application as well, under performance and energy bounds. Further, $C_1$ and $C_2$ parameters can be tuned to reach the desired tradeoff points. Overall, the results with g.721, mpeg and specjbb show that our approach can be augmented with profile data to make it applicable to the cases where the compiler cannot completely extract the UMAG from the source code.

## 7. Conclusion

The main contribution of this paper is a compiler directed NoC reliability enhancement mechanism based on packet duplication. In this approach, the underlying NoC architecture of the CMP is exposed to the compiler, which in turn determines the routes for both original and duplicate messages such that the potential impacts of both transient and permanent link errors could be mitigated. Our approach also tries to satisfy performance and power constraints by, respectively, minimizing the set of common links between an original message and its duplicate and maximizing the link reuse between neighboring program phases. We fully implemented our approach within an optimizing compiler, and the collected results indicate that it is much more effective than an alternate approach to NoC reliability and performs better than two pure hardware based fault tolerance schemes.

## References


[1] M. Ali et al. A Fault Tolerant Mechanism for Handling Permanent and Transient Failures in a Network-on-Chip. In Proc. ITNG, 2007.

[2] AMD Athlon 64 X2 Dual-Core Processor for Desktop. http://www.amd.com /us-en/Processors/ProductInformation/0,,30_118_9485_13041,00.html

[3] J. M. Anderson. Automatic Computation and Data Decomposition for Multiprocessors. Ph.D Thesis, Stanford University, 1997.

[4] G. Ascia et al. Multi-objective Mapping for Mesh-based NoC Architectures. In Proc. CODES+ISSS, 2004.

[5] T. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In Proc. MICRO, 1999.

[6] D. Bertozzi et al. Low Power Error Resilient Encoding for On-Chip Data Buses. In Proc. DATE, 2002.

[7] D. Brooks et al. Wattch: A Framework for Architectural-level Power Analysis and Optimizations, In Proc. ISCA, 2000.

[8] G. Chen et al. Compiler-directed Channel Allocation for Saving Power in On-chip Networks. In Proc. POPL, 2006.

[9] G. Chen et al. Reducing NoC Energy Consumption Through Compiler-Directed Channel Voltage Scaling. In Proc. PLDI, 2006.

[10] K. Coons et al. A Spatial Path Scheduling Algorithm for EDGE Architectures. In Proc. ASPLOS, 2006.

[11] W. J. Dally and B. Towles. Route Packets, Not Wires: On-chip Interconnection Networks. In Proc. DAC, 2001.

[12] G. De Micheli. Reliable Communication in SoCs. In Proc. DAC, 2004.

[13] J. Duato. A New Theory of Deadlock-Free Adaptive Routing in Wormhole Networks. IEEE TPDS 4(12):1320–1331, 1993.

[14] M. W. Hall et al. Maximizing Multiprocessor Performance With the SUIF Compiler. IEEE Computer, December 1996.

[15] Y. Hoskote et al. A 5-GHz Mesh Interconnect for a Teraflops Processor. In IEEE MICRO, Sept/Oct, 2007.

[16] L. Hsu et al. Exploring the Cache Design Space for Large Scale CMPs. In SIGARCH Comput. Archit. News, 33(4):24–33, 2005.

[17] J. Hu and R. Marculescu. Energy- and Performance-Aware Mapping for Regular NoC Architectures. IEEE TCAD, 24(4):551–562, April, 2005.

[18] http://www.intel.com/idf/.

[19] Intel quad-core Xeon. http://www.intel.com/quad-core/?cid=cim: ggl—xeon_us_clovertown—k7449—s

[20] J. Kahle et al. Introduction to the Cell Multiprocessor. IBM Journal of Research and Development, 49(4-5), 2005.

[21] M. Kandemir and O. Ozturk. Software-Directed Combined CPU/Link Voltage Scaling for NoC-Based CMPs. In Proc. SIGMETRICS, 2008.

[22] C. Kim et al. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. In Proc. ASPLOS, 2002.

[23] P. Kongetira et al. Niagara: A 32-Way Multithreaded SPARC Processor. IEEE MICRO, Apr., 2005.

[24] C. Lee et al. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In Proc. MICRO, 1997.

[25] W. Lee et al. Space-Time Scheduling of Instruction-Level Parallelism on a RAW Machine. In *Proc. ASPLOS*, Oct. 1998.

[26] F. Li et al. Profile-Driven Energy Reduction in Network-on-Chips. In Proc. PLDI, San Diego, 2007.

[27] F. Li et al. Compiler-directed Proactive Power Management for Networks. In Proc. CASES, 2005.

[28] R. McGowen. Adaptive Designs for Power and Thermal Optimization. In Proc. ICCAD, 2005.

[29] A. Mejia et al. Segment-Based Routing: An Efficient Fault-Tolerant Routing Algorithm for Meshes and Tori. In Proc. IPDPS, 2006.

[30] S. Murali et al. Analysis of Error Recovery Schemes for Networks on Chips. In IEEE Design and Test, 2005.

[31] R. Nagarajan et al. Static Placement, Dynamic Issue (SPDI) Scheduling for EDGE Architectures. In Proc. PACT, 2004.

[32] E. Oh et al. Fault-Tolerant Routing in Mesh-Connected 2D Tori. In Proc. ICCS, 2003.

[33] M. Pirretti et al. Fault Tolerant Algorithms for Network-on-Chip Interconnect. In Proc. IEEE VLSI, 2004.

[34] V. Soteriou and L.-S. Peh. Design Space Exploration of Power-Aware On/Off Interconnection Networks. In Proc. ICCD, 2004.

[35] SPEC. http://www.spec.org/cpu2000/CINT2000/.

[36] SPEC. http://www.spec.org/jbb2005/

[37] C. C. Su and K. G. Shin. Adaptive Fault-Tolerant Deadlock-Free Routing in Meshes and Hypercubes. IEEE TC, 45(6):666–683, 1996.

[38] D. Tarjan et al. CACTI 4.0. HP Labs, Tech. Rep. HPL-2006-86, 2006.

[39] T. Theocharides et al. Networks on Chip: Interconnects for the Next Generation Systems on Chip. In Advances in Computers, Vol 63, 2005.

[40] Virtutech Simics. http://www.virtutech.com/

[41] H.-S. Wang et al. Orion: A Power-Performance Simulator for Interconnection Networks. In Proc. MICRO, 2002.

[42] J. Wu. Fault-Tolerant Adaptive and Minimal Routing in Mesh-Connected Multicomputers Using Extended Safety Levels. In IEEE TPDS, 11(2):149–159, 2000.

[43] Xpress-MP. http://www.dashoptimization.com/pdf/Mosel1.pdf, 2002.

[44] J. Zhou and F. C. M. Lau. Adaptive Fault-Tolerant Wormhole Routing in 2D Meshes. In Proc. IPDPS, 2001.

[45] X. Zhu and W. Qin. Prototyping a Fault-Tolerant Multiprocessor SoC With Runtime Fault Recovery. In Proc. DAC, 2006.